APPENDIX A - TABLES

Task ID	T_i	$C_{b,i}$
τ_1	6	1
τ_2	7	1
τ_3	21	2

Table (1) - Periodic Task Set

Notation	Description
τ_i	The task in a process with priority level i. In traditional RMA, τ_i is a single thread and the whole system is a single partition. In DEOS, we call τ_i an *aggregate thread.* There are many threads running at the same rate in DEOS. So, if $t_{i,1}, t_{i,2}, \ldots, t_{i,n_i}$ are all the threads defined for rate i, τ_i is the sequence of these threads when run back-to-back. This representation allows us to consider slack only in terms of rates and not in terms of threads which potentially helps performance significantly.
n	The number of distinct (aggregate) threads allowed in the system. This number is fixed at system power up.
T_i	The time between dispatches of τ_i. We assume without loss of generality that $T_1 \leq T_2 \leq \ldots \leq T_n$. T_i is also called the period of τ_i. In DEOS, strict inequality holds.
H	The hyperperiod of the task set. $H = \text{lcm}(T_1, T_2, \ldots T_n)$. Note that in a harmonic system such as DEOS, $H = T_n$.
τ_{ij}	The j^{th} dispatch of τ_i. Again, in DEOS, τ_{ij} is an aggregate thread.
C_{ij}	The worst case execution time for τ_{ij}. In classical RMS the task set is fixed so $C_{ij} = C_i$ for each dispatch j where $j = 1, .., \frac{H}{T_i}$. Note that this quantity is computed at each successful schedulability test.
C_i	A short hand notation for C_{ij} when $C_{ij} = C_{ik}$ for all $j, k \in \{1, \ldots, \frac{H}{T_i}\}$.

Table (2) - Periodic Thread Specification in Classical RMS

Notation	Description
$n_{i/j}$	The value $\frac{T_i}{T_j}$ for $i \geq j$. $n_{i/j}$ is the number of times τ_j will execute during one period defined by T_i. For a harmonic system, all $n_{i/j}$ are integers.
Timeline Slack$_i$	The level i slack in the interval $[0, j \cdot T_i]$ assuming all periodic processes take their worst case execution time to complete.
E_i	The dispatch identifier of the next instance of τ_i to complete. If τ_i is in state CompletedForItsPeriod, this is the next instance, otherwise it is the current instance. This value must be maintained at runtime. When aggregate threads are supported, one state variable per thread may be necessary.
AperiodicTime$_i$	The amount of level i or higher aperiodic time that has been consumed since the beginning of the hyperperiod. This includes all time consumed by aperiodic task of priority $1, ..., i$, where periodic process overrun can be considered aperiodic process computation time. There is an implicit time argument, so AperiodicTime$_i$ = AperiodicTime$_i(t)$.
Idle$_i$	Level i idle time that has occurred since the beginning of the hyperperiod. This is all the time not spent processing tasks of priority i or higher. In other words, it is all the time spent processing tasks (periodic, aperiodic or idle) of priority $i+1, ..., n, n+1$ where n is the number of rates in the system, and level $n+1$ is the idle process. There is an implicit time argument, so Idle$_i$ = Idle$_i(t)$.
γ_i	The dispatch identifier of τ_i, or equivalently the period identifier of T_i. There is an implicit time argument, so $\gamma_i(t) = \gamma_i$.
L_{ij}	The amount of level $i-1$ slack available in $[0, j \cdot T_i]$ which is the amount of time available for processing tasks at level $i-1$ without causing $\tau_1, \tau_2, ..., \tau_i$ to miss any of their deadlines in that interval.

Table (3) - Slack Scheduling Specification in the Context of Classical RMS

Dispatch ID	1	2	3	4	5	6	7
Timeline Slack $_{i,j}$	5 4 12	10 9 25	15 14	20 19	25 24	30 29	35

Table (4) - Timeline Slack $_{i,j}$

Dispatch ID	1	2	3	4	5	6
TimelineSlack(1,30)	4	8	12	16	20	24
TimelineSlack(2,30)	6	12	18			
TimelineSlack(3,30)	10					

Table (5) - TimelineSlack$_{i,j}$

Thread Servics	Description
createThread	Creates a new thread. If the thread is dynamic, it also starts the thread.
startThread	Schedules to have the (static) thread started at the beginning of the next period defined by the threads rate, after the start service has completed.
startThreads	Schedules to have the set of threads started at the beginning of each of their respective periods defined by their rates, after the start threads service has completed.
restartThread	An active thread is restarted from the beginning.
killThread	An active dynamic thread is deactivated. A stopped static thread is also deactivated.
stopThread	This routine is newly added. Static threads must first be stopped before they can be killed.
waitUntilNextPeriod	The calling thread is suspended until the start of its next period where it resumes execution. Other threads queued at a mutex that the calling thread holds will be dequeued.
restartProcess	All the process' threads, mutexes and events are killed. The process' PRIMARY THREAD is restarted.
createMutex	Creates a mutex that can be accessed by multiple threads in the calling thread's process.
lockMutex	The calling thread is granted the lock if the mutex is unlocked and queues if waitok is sc true.
unlockMutex	A thread releases its lock on a mutex and the lock is granted to the highest priority thread (if any) waiting.
resetMutex	All threads queued at the mutex are dequeued (including an executing thread).
waitForEvent	The calling thread is suspended until the event is pulsed.
pulseEvent	All threads currently waiting for the pulsed event will transition from state suspended to state ready.

Table (6) - Thread Services

Notation	Description
τ_i	The aggregate of threads with priority level i. We call τ_i an aggregate thread.
n	The number of distinct rates allowed in the system. This number is fixed between coldstarts.
$t_{i,j}$	The j^{th} thread of priority level i. Even though there is no explicit ordering of threads within a priority level, it is convenient to do so for the sake of reference.
T_i	The time between dispatches of τ_i. We assume without loss of generality that $T_1 < T_2 < \ldots < T_n$.
γ_i	The period identifier. At time t where $t \in [0, H]$, $\gamma_i(t) = \gamma_i = \lceil \frac{t}{H} \rceil$.
$m_i(t)$	The number of active threads forming τ_i at time t. For ease of exposition, the t is often omitted and refers to the current period of T_i so $m_i(t) = m_i$. Note that there is a time lag between thread creation and thread activation.
m_i'	A temporary value for m_i when threads will but have not yet become (de)activated.
C_i	The worst case budget times summed over all threads forming τ_i.
$n_{i\|j}$	The value $\frac{T_i}{T_j}$ for $i \geq j$. $n_{i\|j}$ is the number of times τ_j will execute during one period defined by T_i.
ζ_k	The primary budget of process k, $k \in \{1, \ldots, p\}$, $p =$ number of active processes. Note that a process can be active and have its primary thread stopped, in which case some portion of its budget is available as timeline slack. This is poor notation actually since the set of active processes changes.
$\mathcal{Z}$	The set of all processes whose unallocated primary budgets are available for slack.
ζ	The sum of the ζ_k with budgets available for slack. $\zeta = \sum_{k \in \mathcal{Z}} \zeta_k$.
U_B	System level utilization reserved for blocking times. A feasibility test is always of the form $U \leq 1 - U_B$.

Table (7) - Periodic Thread Notation

Notation	Description
A_i (also Timeline Slack$_i$)	is the amount of timeline slack that was made available from processes with inactive primary thread budgets with rate i at time $\gamma_i(t)T_i$. **Note:** A_i is not cumulative since the beginning of the hyperperiod. Also, in the current release of DEOS, it is always true that $A_j = 0$ for $j \in \{2, ..., n\}$.
A	The vector $(A_1, A_2, ..., A_n)$ which is maintained at run-time.
$\Delta A_{i,j}$	The amount to change rate A_i the next time the start of periods defined by T_j and T_i coincide. It will be the case that for $i > j$, $\Delta A_{ij} = 0$. Values of ΔA_{ij} are updated to reflect user thread (de)activation requests at level i with an inactive primary thread at level j. Note also that if there are no primary threads active at rate i, then $\Delta A_{ij} = 0 \forall j$.
m_i	The number of threads in aggregate thread τ_i for $i = 1, ..., n$. $m_i = m_i(t)$.
$t_{i,k}$	The k^{th} thread in τ_i, for $k = 1, ..., m_i$.
$\beta_{i,k}$	The budget of t_{ik}. Set when t_{ik} is created.
$\xi_{i,k}$	The actual execution time of t_{ik} for the current dispatch. If the current dispatch has completed then it is the total time that dispatch of t_{ik} took to execute. $0 \leq \xi_{ik} \leq \beta_{ik}$.
E_i	A boolean value indicating τ_i's activation status. If τ_i is active, E_i = TRUE otherwise E_i = FALSE. This value is maintained at runtime.
Aperiodic Time$_i$(t)	the amount of level i aperiodic time consumed in $[\gamma_i(t)T_i, t]$. For simplicity, we denote AperiodicTime$_i$(t) = AperiodicTime$_i$.
Idle$_i$(t)	the amount of "level" i idle time (i.e. time spent running the idle process) in $[\gamma_i(t)T_i, t]$ no longer available to slack. For simplicity we denote Idle$_i$(t) = Idle$_i$.
$\mathcal{L}_i(t)$	a conservative estimate of the amount of level i idle time that is lost as level i reclaimed slack due to sitting idle.
$\mathcal{R}_i(t)$	The amount of slack reclaimed by completing for period at level i in $[\gamma_i(t)T_i, t]$.
$\gamma_i(t)$	The period identifier for T_i. For $i \in \{1, 2, ..., n\}$, $\gamma_i(t) = \lfloor \frac{t}{T_i} \rfloor$. Alternatively, one can think of γ_i as the dispatch identifier for τ_i, $\gamma_i \in \{0, 1, ..., H/T_i - 1\}$.
U$_k$	A conservative value of the amount of level k slack available that can be carried over to the next period T$_k$.
CurID(i)	This is associated with the system, and uniquely identifies the period T_i. Comparisons of the form P.ReqID(i) $\leq$ CurID(i) will appear in the algorithms. Sometimes these will be abbreviated $P.\gamma_i < \gamma_i$, where uniqueness is understood. See comments in the text about counter roll over.
USys	System utilization allocated to active processes, including pending requests for creation/activation and deletion/deactivation. Note that USys does not necessarily reflect the current utilization allocated to active processes.
ΔUSys(1..n)	Changes to the actual process utilization allocated to active processes that will take place at the next period boundary of T_i, at level i.
$n^r_{j\|i}(t)$	The remainder of *full* T_j periods remaining in the *current* (relative to t) T_i period. In symbols, $n^r_{j\|i}(t) = \lfloor((\gamma_i(t) + 1)T_i - t)/T_j\rfloor$.
$B^r_j(t)$	The remainder of any unused fixed budgets belonging to ISR threads at rates $1, ..., j$ in the interval $[t, (\gamma_j(t) + 1)T_j]$.
$B^t_j(t)$	The sum total of all fixed budgets belonging to ISR threads at rates $1, ..., j$ in any T_j period. In this release of DEOS, if $B(t)$ is the worst case "aggregate" ISR fixed thread budget (at time t, since ISR threads can be killed/created), $B^t_j(t) = n_{j\|1}B(t)$, a quantity that should be easy to maintain at runtime.

Table (8) - Slack Scheduling Notation

Notation	Description
UserBudget	The total amount of time (normalized by the process' primary thread's period) allocated to active users within the process. UserBudget will never exceed MaxBudget, which is the process' entire budget. UserBudget reflects any pending changes indictated by ΔBudgetReq. Consequently, UserBudget is not necessarily the current value of the process' budget assigned to user thread. But that value can be computed.
MaxBudget	The process' total budget, normalized by the period of its primary thread. The term budget is somewhat misleading. Utilization is a more descriptive term.
Rate	The rate at which the highest priority thread (including the process' primary thread) runs. Note that no user thread of a process p will have a rate higher than the process' primary thread. It is TBD whether there is benefit in having a primary thread with rate higher than any of its users. Rate takes on one of the values $1,\ldots,n$, with 1 the highest rate, and n the slowest rate.
Active	A boolean value set to TRUE when the primary thread is active and false when the primary thread is inactive. When p.Active is FALSE, the primary thread's budget is made available as timeline slack.
ProcActive	A boolean value set to TRUE when the process (not just its primary thread) is active, otherwise it is FALSE. $\neg$ P.ProcActive $\Rightarrow$ $\neg$ P.Active (regardless of its value).
ReqID(i)	This uniquely represents the most recent time a request for user thread (de)activation has been made at level i. **Note:** it is not sufficient to use γ_i since these table values are not updated "periodically", but only when other (de)activations take place after the requests have been processed.
γ_i	We sometimes denote P.ReqID(i) by $P.\gamma_i$ where it is understood that $P.\gamma_i$ uniquely defines the request period T_i.
ΔBudgetReq(i)	This is the amount of change in allocated budget at level i that either will or did occur at time $(\text{ReqID}_i(t) + 1)T_i$. If the change hasn't yet occurred, subsequent requests might change this value.

Table (9) - Process Record Attributes (Budget Update Vector)

Notation	Description
ComputeTime	The total compute time allocated to the thread. A timeout will be enforced to ensure that a thread does not exceed its worst case compute time.
CT	An abbreviation for ComputeTime.
ExecTime	The total time spent executing so far. This time is updated at each thread preemption or suspension.
ET	An abbreviation for ExecTime.
TimeSlice	The amount of time a thread is allowed to execute prior to a hardware timeout. Examples of timeouts are maximum mutex execution times and maximum available slack consumption before thread suspension.
TS	An abbreviation for TimeSlice.
E_i	A boolean, denoted by E_i for aggregate thread τ_i which is true if all threads at rate i have a true value for CompletedForItsPeriod and false otherwise.
ExecutingOnSlack	A boolean value which is true when a thread's budget current budget has been from taken from the slack pool and false when it is a part of its fixed budget.

Table (10) - Thread State Time Variables

Notation	Description
Slack	A record perhaps indexed by slack level (depending on the slack consumption algorithms) containing the amount of slack reclaimed at level i, and the most recent period T_i during which it was reclaimed.
Slack.γ_i	The identifier of the most recent T_i period during which level i slack was consumed. $i \in \{0, ..., H/T_i - 1\}$. This attribute is not used in the maximal update set of algorithms.
Slack.$\mathcal{R}_i$	The amount of slack reclaimed by completing (early) for period at level i within the "current" period defined by γ_i. Slack.$\mathcal{R}_i$ is set to zero at every period boundary defined by T_i.
$\mathcal{R}_i$	An abbreviation for Slack.$\mathcal{R}_i$, which works only when the slack record is not indexed.
Slack.U_i	The amount of unused slack at level i that has been carried forward at time $\gamma_i(t)T_i$. Slack.U_i is recalculated at every period boundary defined by T_i.
U_i	An abbreviation for Slack.U_i, which again works only when the slack record is not indexed.
Slack(j)	The slack record associated with a slack consuming thread (if any) at level j. In this situation, slack made available at the higher rates is allocated directly to high rate slack consumers, without taking away (or recalculating) slack previously allocated to low rate slack consumers. This record is not used in the maximal update set of algorithms.

Table (11) - Slack Record Attributes

APPENDIX B - ALGORITHMS

```
-- Algorithm UpdateIdleSlackVariables($i$: in priority);
-- This algorithm updates the idle slack variables used when computing slack availability.
-- It is called whenever a periodic task completes.
-- update_time = the worst case time to execute this routine, a constant (perhaps based on $i$).

  $E_i := (E_i + 1)\text{mod}\frac{H}{T_i}$; - update the activation status
  idle_time_consumed := execution_time($\tau_i$);
  slack_reclaimed := worst_case_execution_time($\tau_i$) - idle_time_consumed;
  for $j := 1, ..., i - 1$ loop
     $\mathcal{I}_j := \mathcal{I}_j$ + idle_consumed + update_time;
  end loop;
  for $j := 1, ..., n$ loop
     $\mathcal{I}_j := \mathcal{I}_j$ - slack_reclaimed + update_time;
  end loop;
```

Algorithm (1) – Update Idle Slack Variables

```
-- Algorithm UpdateAperiodicSlackVariables($i$: in priority, $t$ : slack consuming thread);
-- This algorithm updates the aperiodic slack variables used when computing slack availability.
-- It is called whenever an aperiodic task completes, which might include surplus compute time for a periodic
-- task increment, or the idle task completing.
-- update_time = the worst case time to execute this routine, a constant (perhaps dependent on $i$).

  -- the aperiodic task $t$ may execute over several time increments. i.e. it may be scheduled,
  -- consume all slack, suspend itself, be rescheduled when more slack is available, etc.
  slack_consumed := execution_time_since_last_scheduling($t$);
  for $j := 1, ..., i - 1$ loop
     $\mathcal{I}_j := \mathcal{I}_j$ + slack_consumed + update_time;
  end loop;
  for $j := i, ..., n$ loop
     $\mathcal{I}_j := \mathcal{I}_j$ + update_time;
     $\mathcal{A}_j := \mathcal{A}_j$ + slack_consumed;
  end loop;
```

Algorithm (2) – Update Aperiodic Slack Variables

```
-- Function AvailableSlack(i: in priority) return time;
-- This algorithm calculates the slack available beginning at the time of the call and ending at the
-- end of the period defined by $T_i$, assuming no new threads were created and no existing threads are destroyed.

    slack_calc_time := the worst case time to execute this procedure (perhaps based on i).
    for $j := 1, ..., i - 1$ loop
       $\mathcal{I}_j := \mathcal{I}_j +$ slack_calc_time;
    end loop;
    for $j := i, ..., n$ loop
       $\mathcal{I}_j := \mathcal{I}_j +$ slack_calc_time;
       $\mathcal{S}_j := A_{j,E_j} - (\mathcal{A}_j + \mathcal{I}_j)$;
    end loop;
    slack_available := $\min_{i \leq j \leq n} \mathcal{S}_j$;
    return slack_available;
    -- in practice, return zero if slack_available < cswap time $+\delta$;
    -- updateAperiodicSlackVariables should be called prior to execution of this routine, if necessary.
```

Algorithm (3) – Available Slack

```
Indefinite Timeout Protocol(c : caller; r : resource);

   if not_available(r) then
      if c.has_successors
         then c.state := CompleteForPeriod;
            reclaim_slack(c.remaining_FixedBudget);
            dequeue(c,queue(r));
         else
            if c.ExecutingOnSlack then c.budget_remaining := available_slack(c.rate); end if;
            if c.budget_remaining < queue_time(c)
               then c.state := CompletedForPeriod;
                  -- slack reclamation may not be worth it here.
               else
                  enqueue(c,queue(r));
                  if c.Slack and SlackOn
                     then reclaim_slack(c.remaining_FixedBudget - c.resource_time(r));
                        c.state := PassivelyWaitingForEvent;
                        Predecrement slack accumulators by c.resource_time(r);
                     else
                        c.state := ActivelyWaitingForEvent;
                        -- This type of wait introduces effective blocking.
                  end if;
            end if;
      end if;
   else
      if c.ExecutingOnSlack then c.budget_remaining := available_slack(c.rate); end if;
      if c.budget_remaining < resource_time(r)
         then c.state := CompletedForPeriod;
            release(r); dequeue(c,queue(r));
         else continue the execution of c with resource r available;
            -- Slack accumulators need to be predecremented if c is executing on slack and r is a mutex.
      end if;
   end if;
```

Algorithm (4) – Indefinite Timeout Protocol for a Resource Wait

```
Long Duration Timeout Protocol(c : caller; r : resource; num_iter : natural);

 if not_available(r) then
    if num_iter = max_iter
       then dequeue(c,queue(r); return;
       else num_iter := num_iter + 1;
    end if;
    if c.has_successors
       then c.state := CompleteForPeriod;
          reclaim_slack(c.remaining_FixedBudget);
          dequeue(c,r); - At the next start of c's next period, c will be moved to r's queue by DEOS
       else
          if c.ExecutingOnSlack then c.budget_remaining := available_slack(c.rate); end if;
          if c.budget_remaining < queue_time(c)
             then c.state := CompletedForPeriod;
                -- slack reclamation may not be worth it here
             else
                enqueue(c,queue(r));
                if c.Slack and SlackOn
                   then reclaim_slack(c.remaining_FixedBudget - c.resource_time(r));
                      c.state := PassivelyWaitingForEvent;
                      Predecrement slack accumulators by c.resource_time(r);
                   else
                      c.state := ActivelyWaitingForEvent;
                      -- This type of wait introduces effective blocking.
                end if;
          end if;
    end if;
 else
    if c.ExecutingOnSlack then c.budget_remaining := available_slack(c.rate); end if;
    if c.budget_remaining < resource_time(r)
       then c.state := CompletedForPeriod;
          release(r); dequeue(c,queue(r));
       else continue the execution of c with resource r available;
          -- Slack accumulators need to be predecremented if c is executing on slack and r is a mutex.
    end if;
 end if;
```

Algorithm (5) – Long Duration Timeout Protocol for a Resource Wait

```
Short Duration Timeout Protocol($c$ : caller; $r$ : resource);

    while $c$.priority <= ready_thread.(inherited)priority
        wait; -- higher or equal priority threads are running
    end while;
    -- $c$ is at the head of queue($r$)
    if not_available($r$)
        then
            return timeout status to $c$; dequeue($c$,queue($r$));
            $c$.state := CompletedForPeriod;
        else
            case $r$.type is
                when $r$ = event => continue the execution of $c$ with event $r$ pulsed;
                when $r$ = mutex => grant $c$ the lock to mutex $r$;
                    -- when $c$ is executing on slack, predecrement the slack accumulators
                when $r$ = semaphore => $c$ calls wait($r$);
            end case;
    end if;
```

Algorithm (6) – Short Duration Timeout Protocol for a Resource Wait

```
-- Algorithm SystemInitializationOfSlackVariables;
-- This algorithm is called once before the start of the initial hyperperiod.
-- Failure to initialize slack variables at this time will result in bogus values later.
-- This algorithm requires modifications when primary thread periods can be other than $T_1$.


    $\mathcal{Z} := \emptyset$;
    $\zeta := 0$;  $\zeta_0 := 0$;
    for each process $P$ in the registry loop
        calculate/read $P$'s budget, $\zeta_p$;
        P.UserBudget := 0; P.MaxBudget := $\zeta_p$; P.Rate := 1;
        P.ReqID := $(0,0,...,0)$; P.$\Delta$BudgetReq := $(0,0,...,0)$;        -- vectors of $n$ zeroes.
        -- Most likey, if P.ProcActive, then P.Active will be assumed at system startup?
        if $P$.ProcActive then
            if P.Active then            --P's primary_thread is active
                then $\zeta_0 := \zeta_0 + \zeta_p$;
                else $\zeta := \zeta + \zeta_p$; $\mathcal{Z} := \mathcal{Z} \cup \{P\}$;
            end if;
        end if;
    end loop;

    for $i := 1,...,n$ loop
        $T_i$ := the period of the $i^{th}$ smallest rate declared in the system;
        $A_i := 0; C_i$ := execution time of $\tau_i$;
        $\Delta$Usys(i) := 0;
        for $j := 1,...,i$ loop
            $n_{i|j} := T_i/T_j$;  $\Delta A_{i,j} := 0$;
            -- $(n_{i|j})$, $(\Delta A_{i,j})$ are diagonal matrices.
        end loop;
    end loop;
    $A_1 := T_1 - (\zeta + \zeta_0 + U_B)$;
    --  $A_1$ := system level slack = budget not assigned to active processes minus system blocking time;
    USys := $(\zeta + \zeta_0)/T_1$;
```

Algorithm (7) – System Initialization of Slack Variables

```
-- Algorithm PeriodUpdateOfSlackVariables($j$ : in rate);
-- This algorithm is called at the start of every period. That is at times $0, T_j, 2T_j, \ldots$

-- This algorithm is called once at the largest period. That is, the start of a period for $T_j$
-- is also the start of a period for $T_k$ when $k \leq j$.

-- $r$ indexes the rates at which primary periods are supported.
-- In the current release of DEOS, $r = 1$, always, so this is an $O(n)$ routine.

    -- When thread (de)activations occured, update changes in dynamic period timeline slack.
    -- One may have to introduce process sets with indices for their primary threads.

    $\mathcal{Z} := \mathcal{Z}'$;  $\mathcal{Z}' := \mathcal{Z}$;
    for $k := 1..j$ loop
        $U_k$ is a conservative amount of level $k$ slack available and not used in the last $T_k$ period that can be carried over.
        Not all of $\mathcal{R}_k$ can be attributed to $U_k$ but can safely be assigned to $U_n$.
        $U_k := U_k + \max(0, (\mathrm{A}_k - (\mathcal{A}_k + \mathcal{I}_k))$;
        $U_n := U_n + \mathcal{R}_k$;
        $\mathcal{I}_k := 0; \mathcal{A}_k := 0; \mathcal{R}_k := 0; \mathcal{L}_k := 0$;
        $E_k :=$ FALSE; $\gamma_k := \gamma_k + 1$;
        $\Delta\mathrm{Usys}(k) := 0$;
        $\sigma := 0$;
        -- In this release of DEOS, the loop here is simply $A_k := A_k + \sum_{r=1}^{j} \Delta A_{1r}$;
        -- and then zero out the $\Delta A_{1r}$ entries.
        for $r := k..j$ loop
            $\sigma := \sigma + \Delta\mathrm{A}_{kr}$;
            $\Delta\mathrm{A}_{kr} := 0$;
        end loop;
        $\mathrm{A}_k := \mathrm{A}_k + \sigma$;
    end loop;

    if $j = n$ then                -- we are at the hyperperiod, reset the period id's and unconsumed slack.
        for $k := 1..n$ loop
            $\gamma_k := 0$;
            $U_k := 0$;
        end loop;
    end if;
```

Algorithm (8) – Period Update of Slack Variables

```
-- Algorithm PrimaryThread(De)activation($P$ : process);
-- If $P$.active, then deactivate $P$'s primary thread else activate $P$'s primary thread.
-- (De)activation request "processing" time is at $s$, where $\gamma_r T_r \leq s < (\gamma_r + 1)T_r$, with $r = P$.rate.

   -- Notation used defined below and is the same as that above.
   --  $n_{j|r} = T_j/T_r$.
   $r := P$.rate;
   -- There may be some pending requests for activations/deactivations that will not be in effect at time $(\gamma_r(s) + 1)T_r$.
   for $j := r..n$ loop
      if P.$\Delta$BudgetReq(j) > 0 and then CurID(j) > P.ReqID(j) then
         -- These updates have already occurred. Zero them out.
         P.$\Delta$BudgetReq(j) := 0; P.ReqID(j) := 0;
      end if;
      if (CurID(j) = P.ReqID(j) and $((\gamma_j + 1)T_j > (\gamma_r + 1)T_r)$) then
         P.UserBudget := P.UserBudget $-$ $\Delta$BudgetReq($j$)/$n_{j|r}$;
         -- $\Delta$BudgetReq($j$) is left unchanged for later updates.
      end if;
      if P.Active then
         $\Delta A_{rr} := \Delta A_{rr} -$ (P.MaxBudget $-$ P.UserBudget)$T_r$;
      else;
         $\Delta A_{rr} := \Delta A_{rr} +$ (P.MaxBudget $-$ P.UserBudget)$T_r$;
      end if;
   end loop;
   if P.Active then P.Active := FALSE else P.Active := TRUE; end if;
```

Algorithm (9) – Updates for Primary Thread (De)Activation Requests

```
-- Algorithm UserThread(De)Activation($\delta$: time; $j$ rate; $P$: process; activation : boolean);
-- $P$ is the process of the thread being (de)activated.
-- $j$ is the rate of the thread for which (de)activation is being requested.
-- $\delta$ is $+/-$ the budget of the thread (in time, not utilization) requesting de/activation.
-- I.e. $\delta < 0$ the call is for a deactivation request, and $\delta > 0$ the call is for an activation request.
-- activation is a boolean, which is actually redundant information given $\delta$'s sign.
-- Can we admit a new thread at level $j$?
-- (De)activation request time is at $s$, where $\gamma_j T_j \leq s < (\gamma_j + 1)T_j$.
-- This assumes that deactivations have at most a one period delay, which is coming soon.
-- The execution of this code is at time $s$, with period boundary code executed at time $(\gamma_j + 1)T_j$.


  -- Notation used defined below.
  --   $n_{j|h} = T_j/T_h$.

  $r$ := $P$.rate;                     --P's primary thread's rate.
  -- CurID($j$) is similar to $\gamma_j$, except it must uniquely identify which period $T_j$ we are in since
  -- $P$.CurID might not have been updated for many hyperperiods, hours, or since system boot.
  for $i$ := 1,..,$n$ loop
     if $P$.$\Delta$BudgetReq(j) $\neq$ 0 then
        if CurID($i$) > $P$.ReqID($i$) or P.ReqID(i) - CurID(i) > $n_{n|1}$ then
           $P$.ReqID($i$) := 0; $P$.$\Delta$BudgetReq($i$) := 0;
           -- These updates have already been made. Zero them out.
        end if;
     end if;
  end loop;

  -- If an activation check for feasibility, and if feasible readjust the compute times within the process.
  if activate then
     UserBudget :=P.UserBudget;
     for $i$ := $j$ + 1..$n$ loop
        UserBudget := UserBudget − min(0,P.$\Delta$BudgetReq(i));
     end loop;
     if UserBudget $+\delta/T_j$ > $P$.MaxBudget
        then
           reject the activation request on the grounds of infeasibility;
           return;
     end if;
  end if;

  $P$.UserBudget := $P$.UserBudget + $\delta/T_j$;
  $P$.ReqID($i$) := $P$.CurID($i$); $P$.$\Delta$BudgetReq := $P$.$\Delta$BudgetReq + $\delta/T_j$;
  if not $P$.Active then
     $\Delta A_{r,j}$ := $\Delta A_{r,j}$ + $\delta/n_{j|r}$;
     -- Here is where we would update $\Delta C_{rj}$ if aggregate threads are used.
  end if;
```

Algorithm (10) – Updates for User Thread (De)Activation Requests

```
-- Algorithm Process(De)activation($P$ : process, $r$ : rate; activate : boolean);
-- This request is made at time $t$. If granted, it will become effective at time $(\gamma_r(t) + 1)T_r$ where $r := P$.Rate;


 $\zeta_p$ := the worst case compute time of $P$ measured every $T_r$ time units. (This would be input in a create.)
 if activate
   then
     if $P$.ProcActive then return either an with error or as a no-op; end if;          -- $P$ is already active.
     -- Determine whether activating P will result in a feasible process set.
     SysBud := SysU;
     for $i := r + 1, ..., n$ loop
        SysBud := SysBud - min(0, $\Delta$Sys(i));
     end loop;
     if SysBud $+\zeta_p/T_r > 1 - U_B$ then
        reject activation request; return;                -- infeasible
     end if;
     -- Activation is feasible
     P.Rate := $r$; P.Active := TRUE; P.ProcActive := TRUE;
     P.MaxBudget := $\zeta_p$; P.UserBudget := 0;
     PrimaryThreadActivation(P);
     $\Delta$Sys(r) := $\Delta$Sys(r) + $\zeta_p/T_r$;
   else            a deactivation request which have no feasibility test
     if P.UserBudget $\neq$ 0 then return error; end if;
     PrimaryThreadDeactivation(P);
     P.ProcActive := FALSE;
     $\Delta$Sys(r) := $\Delta$Sys(r) - $\zeta_p/T_r$;
 end if-then-else;

 -- This is another place where the $\Delta$C matrix is updated, and also $\mathcal{Z}'$.
 --       The latter may not be needed when all processes are assumed to be declared statically.
```

Algorithm (11) – Process (De)Activation

```
-- Algorithm UpdateReclaimedSlack($i$: in priority);
-- This algorithm updates the reclaimed slack variables used when computing slack availability.
-- It is called whenever a task executing on fixed budget completes for period.
-- The same algorithm applies whether doing incremental or aggregate updates.


   if $\tau_i$ has completed then
      slack_reclaimed := ComputeTime($\tau_i$)- ExecTime($\tau_i$) - update_time;
      -- update_time is the time to execute this code
      $\mathcal{R}_i := \mathcal{R}_i$+ max(0,slack_reclaimed - $\mathcal{L}_i$);
   end if;
```

Algorithm (12) – Update Reclaimed Slack Variables

```
-- Algorithm UpdateIdleSlackVariables;
-- This algorithm updates the idle slack variables used when transitioning from busy to idle.
-- It is called whenever when the idle task completes (at priority (n+1)).
-- I.e. the idle process is denoted by $\tau_{n+1}$.
-- update_time = the worst case time to execute this routine.


  idle_time := ExecTime($\tau_{n+1}$);
  -- The assumption for DEOS is that idle_time $\leq T_1$.
  -- To relax that assumption would require more bookkeeping.
  $j := 1$;
  while idle_time > 0 and $j \leq n$ loop
     -- slack_available := $(A_j + U_j + R_j) - (A_j + I_j)$
     if idle_time $\leq$ slack_available then
        $\mathcal{I}_j := \mathcal{I}_j +$ idle_time; idle_time := 0;
     end if;
     -- $C_j$ is defined as the worst case compute time, but can be reduced
     -- to the worst case amount of time threads at level $j$ can wait for a resource
     -- and not give up their time to the slack.
     -- In DEOS, as I understand it, this is only ISR threads which run at rate 1.
     -- I believe the algorithm works for threads that can wait for resources while holding budgets.
     -- I also think under these conditions, it is suboptimal.
     if slack_available < idle_time and idle_time $\leq (A_j + U_j + C_j)$ then
        $\mathcal{I}_j := \mathcal{I}_j +$ slack_available;
        $\mathcal{L}_j := \mathcal{L}_j +$ (idle_time - slack_available);
        idle_time := 0;
     end if;
     if idle_time > $(A_j + U_j + C_j)$ then
        $\mathcal{I}_j := \mathcal{I}_j +$ slack_available;
        $\mathcal{L}_j := \mathcal{L}_j + (A_j + U_j + C_j) -$ slack_available;
        idle_time := idle_time - $(A_j + U_j + C_j)$;
     end if;
  end loop;
```

Algorithm (13) – Update Idle Slack Variables

```
-- Algorithm UpdateAperiodicSlackVariables($i$: in priority, $t$ : slack consuming thread);
-- This algorithm updates the aperiodic slack variables used when computing slack availability.
-- It is called whenever an aperiodic task completes, which might include surplus compute time for a periodic
-- task increment, or the idle task completing.
-- update_time = the worst case time to execute this routine, a constant (perhaps dependent on $i$).


-- the aperiodic task $t$ may execute over several time increments. i.e. it may be scheduled,
-- consume all slack, suspend itself, be rescheduled when more slack is available, etc.
slack_consumed := execution_time_since_last_scheduling($t$) + update_time;
$j := 1$;
while slack_consumed > 0 loop
   ljs := min(slack_consumed, max($(A_j + \mathcal{R}_j + U_j) - (\mathcal{I}_j + \mathcal{A}_j +$ slack_consumed),0));
   slack_consumed := slack_consumed − ljs;
   $\mathcal{A}_j := \mathcal{A}_j +$ ljs;
   $j := j + 1$;
end while;
```

Algorithm (14) – Update Aperiodic Slack Variables

-- Function AvailableSlack return an n-vector of slack time = $(\mathcal{S}(1), \mathcal{S}(2), ..., \mathcal{S}(n))$;
-- This algorithm calculates the slack available beginning at the time of the call, say s and ending at the
-- ends of periods defined by $((\gamma_1(s)+1)T_1, (\gamma_2(s)+1)T_2, ..., (\gamma_n(s)+1)T_n)$.
-- Note that more period timeline slack may become available in these intervals after this request.
-- This differs significantly from the original slack stealer.
-- Note the difference in fonts for A_i and $\mathcal{A}_i$. They are different variables.

```
slack_calc_time := the worst case time to execute this procedure;
$S_U := 0$;
for $j := 1..n$ loop
    $S := (A_j + \mathcal{R}_j + U_j) - (\mathcal{A}_j + \mathcal{I}_j +$ slack_calc_time);
    $S_U := S_U + S$;
    if $S_U < 2(\mathrm{cswap} + \delta) + \mathrm{cachebonus}$
        then $\mathcal{S}_j := 0$;
        else $\mathcal{S}_j := S_U$;
    end if;
end loop;
return $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_n)$;
```

-- in practice, if the slack available at any level is too small to cover the cost of context swaps plus other overhead, using it causes a negative effect.
-- δ and cacheBonus is selected based on system overheads beyond cswaps.
-- UpdateAperiodicSlackVariables should be called prior to execution of this routine, when necessary.
-- UpdateIdleSlackVariables will have automatically be called prior to exection of this routine.

Algorithm (15) – Available Slack